New Book
Elements of Compiler Design
by
Alexander Meduna

Taylor and Francis Group/ Auerbach Publications
New York, 2007
ISBN: 978-1-4200-6323-3
http://www.fit.vutbr.cz/~meduna/books/eocd
Professor Alexander Meduna, PhD

Department of Information Systems
Faculty of Information Technology
Brno University of Technology
Božetěchova 2, Brno, the Czech Republic

E-Mail: meduna@fit.vutbr.cz
Phone: +420 54114-1232
Fax: +420 54114-1270
Website: http://www.fit.vutbr.cz/~meduna
Subject

Approach
- introductory level
- both theoretical and practical treatment

Pedagogical Goals
- understanding compiler design in theory
- learning how to write a compiler in practice

Keywords
- compiler writing
- lexical analysis
- syntax analysis
- syntax-directed translation
- optimization
- code generation
- automata theory
- formal languages
Courses

Primary course
- one-term introductory course in compiler design at an undergraduate level

Secondary course
- automata theory and formal languages
Theoretical aspects of this book

- formal models underlying compilation phases
- formalization of the concepts, methods, and techniques employed in compilers
- mathematical foundations of compilation
- formal languages, grammars, automata, and transducers
Practical aspects of this book

- implementation of compilation techniques
- case study that designs a Pascal-like programming language and its compiler
- many examples and programs
- description of software tools, including yacc and lex
**Features and Their Benefits 1/2**

- **feature:** presents the essentials of compiler writing in an easy-to-follow way
  - **benefit:** students grasp compiler construction quickly and clearly

- **feature:** includes intuitive explanations of theoretical concepts, definitions, algorithms, and compilation techniques
  - **benefit:** students easily follow the topics under discussion

- **feature:** examines the mathematical foundations of compiler design and related topics, such as formal languages, automata, and transducers
  - **benefit:** demonstrates compilation techniques precisely
**Features and Their Benefits 2/2**

- **feature:** demonstrates how theory and practice work together in a real-world context through the implementation of algorithms, examples, case studies, and software tools, such as lex and yacc
  - **benefit:** enhances comprehension

- **feature:** contains the C++ implementation of a real compiler as well as a variety of programs and challenging exercises, many of which are instructively solved
  - **benefit:** demonstrates how to write programs to implement the compilation algorithms

- **feature:** accompanying website provides lecture notes, teaching tips, homework assignments, errata, exams, solutions, and implementation of compilers
  - **benefit:** enhances comprehension
Brief Contents

- Preface (14 pages)
- Introduction (20 pages)
- Lexical Analysis (54 pages)
- Syntax Analysis (64 pages)
- Deterministic Top-Down Parsing (20 pages)
- Deterministic Bottom-Up Parsing (26 pages)
- Syntax-Directed Translation and Intermediate Code Generation (28 pages)
- Optimization and Target Code Generation (20 pages)
- Conclusion (6 pages)
- Appendix (16 pages)
- Bibliography (22 pages)
- Indices (10 pages)
Preface

Introduction
- Mathematical Preliminaries
- Compilation
- Rewriting Systems
Lexical Analysis
- Models
- Methods
- Theory

Syntax Analysis
- Models
- Methods
- Theory
Deterministic Top-Down Parsing
- Predictive Sets and LL Grammars
- Predictive Parsing

Deterministic Bottom-Up Parsing
- Precedence Parsing
- LR Parsing
Syntax-directed Translation and Intermediate Code Generation

- Bottom-Up Syntax-Directed Translation and Intermediate Code Generation
- Top-Down Syntax-Directed Translation
- Semantic Analysis
- Symbol Table
- Software Tools for Syntax-Directed Translation
Contents 5/5

Optimization and Target Code Generation
- Tracking the Use of Variables
- Optimization of Intermediate Code
- Optimization and Generation of Target Code

Conclusion

Appendix: Implementation

Bibliography

Indices

How this book differs
- too complicated for the undergraduate students

Strength
- a revised and updated version of the famous “Dragon Book.”
- covers all the major topics in compiler design in depth
- used as the basis of a graduate class on compilers

Weakness
- written in somewhat dry and encyclopedic way
Competition 2/5


How this book differs
- concentrates its attentions only on the back end of a compiler
- cannot be used at an undergraduate level

Strength
- has a nice layout and gives many examples
- all topics are well connected to each other
- helpful for an advanced computer programmer

Weakness
- avoids any mathematical formalism and theoretical concepts
- text is wordy
Competition 3/5


**How this book differs**
- beyond the level of bachelor students
- necessary to supplement this book, such as Chapter 3 about attribute grammars, with other books on compilers

**Strength**
- covers a broad range of concepts used in modern compilers
- explains the compilation of object-oriented, functional, logic, parallel, and distributed languages
- describes the implementation of optimization techniques in detail

**Weakness**
- algorithms are written in a difficult-to-follow pseudo-code
- exercises at the end of each chapter are rather poor
Competition 4/5


How this book differs
- describes all formal notions in a very informal way
- difficult to understand how these notions are related to the process of compilation

Strength
- provides a throughout introduction to compiler design
- contains all the essential material concerning compilers

Weakness
- presents all concepts in an obscure way
- reader can hardly grasp the principles of compiler writing
- examples are too trivial and somewhat dated
- contains many minor mistakes and misprints

How this book differs
- beyond the level of bachelor students

Strength
- approaches to writing compilers by using C
- includes numerous programs
- covers many advanced topics concerning code generation, optimization, and real-world parsing
- good reference

Weakness
- necessary to supplement this book with books on automata
**Operations REDUCE and SHIFT**

- In a $G$-based bottom-up parser, where $G = (\Sigma, \mathcal{R})$ is a grammar, we use two operations, **REDUCE** and **SHIFT**, which modify the current $pd$ top as follows:
  
  - **REDUCE**$(A \to x)$ makes a reduction according to $A \to x \in \mathcal{R}$
  - **SHIFT** pushes $ins$ onto $pd$ and advances to the next input symbol

**Algorithm 5.2 Operator Precedence Parser**

**Input**

- a grammar $G = (\Sigma, R)$
- a $G$-op-table
- $ins = \uparrow w$ with $w \in \Delta^*$

**Output**

- ACCEPT if $w \in \mathcal{L}(G)$, and REJECT if $w \notin \mathcal{L}(G)$
Method
begin
  set pd to u;
repeat
  case \textit{G-op-table} [pd-top-terminal, ins\textsubscript{1}] of
    | : SHIFT;
    \textbf{\textbackslash l} : SHIFT;
    \textbf{\textbackslash r} : if \textit{G} contains a rule \( A \to x \) with \( x = \textit{G-op-handle} \) then REDUCE\((A \to x)\);
    else REJECT; \{no rule to reduce by\}
    \textbf{\textbackslash u} : REJECT; \{\textit{G-op-table}-detected error\}
    \textbf{\textbackslash a} : ACCEPT;
  end; \{case\}
until ACCEPT or REJECT;
end.
**Case Study**

\[
C \rightarrow C \vee C \\
C \rightarrow C \wedge C \\
C \rightarrow (C) \\
C \rightarrow i
\]

**Operator Precedence Table**
## Operator Precedence Parsing

### Configuration | Table Entry | Parsing Action
--- | --- | ---
1. $\cdot i \land (i \lor i)$ | $[\downarrow, i] = \downarrow$ | SHIFT
2. $i \cdot \land (i \lor i)$ | $[i, \land] = \downarrow$ | REDUCE($C \rightarrow i$)
3. $C \cdot \land (i \lor i)$ | $[\downarrow, \land] = \downarrow$ | SHIFT
4. $C \triangle \cdot (i \lor i)$ | $[(\land, \vert) = \downarrow$ | SHIFT
5. $C \cdot (\triangle i \lor i)$ | $[(, i] = \downarrow$ | SHIFT
6. $C \cdot (i \triangle i \lor i)$ | $[i, \lor] = \downarrow$ | REDUCE($C \rightarrow i$)
7. $C \cdot (C \triangle i \lor i)$ | $[(, \lor] = \downarrow$ | SHIFT
8. $C \cdot (C \lor \triangle i)$ | $[\lor, i] = \downarrow$ | SHIFT
9. $C \cdot (C \lor i \triangle)$ | $[i, \vert] = \downarrow$ | REDUCE($C \rightarrow i$)
10. $C \cdot (C \lor C \triangle)$ | $[(\lor, \vert] = \downarrow$ | REDUCE($C \rightarrow C \lor C$)
11. $C \cdot (C \triangle)$ | $[(, \vert] = \downarrow$ | SHIFT
12. $C \cdot (C \triangle)$ | $[\land, \vert] = \downarrow$ | REDUCE($C \rightarrow (C)$)
13. $C \triangle C \cdot \triangle$ | $[\land, \land] = \downarrow$ | REDUCE($C \rightarrow C \land C$)
14. $C \cdot \triangle$ | $[\land, \land] = \downarrow$ | ACCEPT
### A Sample: Precedence Parsing 5/10

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Rule</th>
<th>Parse Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶️ ◗ i ∧ (i ∨ i) ◗</td>
<td>C → i</td>
<td>$C(\overset{\text{i}}{i}) \wedge (i \lor i)$</td>
</tr>
<tr>
<td>▶️ ◗ i ∧ (i ∨ i) ◗</td>
<td>C → i</td>
<td>$C(\overset{\text{i}}{i}) \wedge (\overset{\text{i}}{i} \lor i)$</td>
</tr>
<tr>
<td>▶️ ♦ (i ∨ i) ♦</td>
<td>C → i</td>
<td>$C(\overset{\text{i}}{i}) \wedge (\overset{\text{i}}{i} \lor i)$</td>
</tr>
<tr>
<td>▶️ C△ ♦ (i ∨ i) ◗</td>
<td>C → i</td>
<td>$C(\overset{\text{i}}{i}) \wedge (\overset{\text{i}}{i} \lor i)$</td>
</tr>
<tr>
<td>▶️ C∧ ♦ (i ∨ i) ◗</td>
<td>C → i</td>
<td>$C(\overset{\text{i}}{i}) \wedge (\overset{\text{i}}{i} \lor i)$</td>
</tr>
<tr>
<td>▶️ C∧ (C∧ ♦) ♦</td>
<td>C → i</td>
<td>$C(\overset{\text{i}}{i}) \wedge (C(\overset{\text{i}}{i} \lor i))$</td>
</tr>
<tr>
<td>▶️ C∧ (C∨ ♦) ♦</td>
<td>C → C ∨ C</td>
<td>$C(\overset{\text{i}}{i}) \wedge (C(\overset{\text{i}}{i} \lor i))$</td>
</tr>
<tr>
<td>▶️ C∧ (C∧ ♦) ♦</td>
<td>C → (C)</td>
<td>$C(\overset{\text{i}}{i}) \wedge C(\overset{\text{i}}{i} \lor i))$</td>
</tr>
<tr>
<td>▶️ C△ C♦ ♦</td>
<td>C → C ∧ C</td>
<td>$C(\overset{\text{i}}{i}) \wedge C(\overset{\text{i}}{i} \lor i))$</td>
</tr>
<tr>
<td>▶️ C♦ ♦</td>
<td>C → C</td>
<td>$C(\overset{\text{i}}{i}) \wedge C(\overset{\text{i}}{i} \lor i))$</td>
</tr>
</tbody>
</table>

**Construction of Parse Tree by Operator-Precedence Parser**
Construction of an Operator Precedence Table

I. if $a$ is an operator that has a higher mathematical precedence than operator $b$, then $a \triangleright b$ and $b \triangleright a$

II. if $a$ and $b$ are left-associative operators of the same precedence, then $a \triangleright b$ and $b \triangleright a$
   if $a$ and $b$ are right-associative operators of the same precedence, then $a \triangleright b$ and $b \triangleright a$

III. if $a$ can legally precede operand $i$, then $a \triangleright i$
     if $a$ can legally follow $i$, then $i \triangleright a$

IV. if $a$ can legally precede $(, then $a \triangleright($
     if $a$ can legally follow $(, then $( \triangleright a$
     if $a$ can legally precede $)$, then $a \triangleright )$
     if $a$ can legally follow $)$, then $) \triangleright a$
A Sample: Precedence Parsing 7/10

Precedence Table with Error-Recovery Routines
Table-Detected Errors

① configuration: \( pd_1 = i \) and \( ins_1 = i \)
   diagnostic: missing operator between two \( is \)
   recovery: change \( pd_1 \) to \( C \), then push \( \wedge \) onto the \( pd \) top

② configuration: \( pd_1 = i \) and \( ins_1 = ( \)
   diagnostic: missing operator between \( i \) and \( ( \)
   recovery: change \( pd_1 \) to \( C \), then push \( \wedge \) onto the \( pd \) top

...
Reduction Errors

1 configuration: \( pd_1 = ( \text{ and } ins_1 = ) \)
   diagnostic: no expression between parentheses
   recovery: push \( C \) onto the \( pd \) top

2 configuration: \( pd_1 \in \{ \land, \lor \} \) and \( ins_1 \notin \{ i, ( ) \} \)
   diagnostic: missing right operand
   recovery: push \( C \) onto the \( pd \) top

...
A Sample: Precedence Parsing 10/10

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Table E.</th>
<th>Parsing Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>▲ ◆ i(∩v)▼</td>
<td>▲, [ =</td>
<td>SHIFT</td>
</tr>
<tr>
<td>▲ ◆ i(∩v)▼</td>
<td>[i, ( = ② table-detected error and rec. ②</td>
<td></td>
</tr>
<tr>
<td>▲ C ∧ ◆ i(∩v)▼</td>
<td>∧, ( =</td>
<td>SHIFT</td>
</tr>
<tr>
<td>▲ ( C ∧ ◆ i(∩v)▼</td>
<td>(, [ =</td>
<td>SHIFT</td>
</tr>
<tr>
<td>▲ C ∧ i(∩v)▼</td>
<td>[i, ∨ ] =</td>
<td>REDUCE(C → i)</td>
</tr>
<tr>
<td>▲ ( C ∧ ◆ i(∩v)▼</td>
<td>(, ∨ ] =</td>
<td>SHIFT</td>
</tr>
<tr>
<td>▲ C ∧ ( C ∧ ◆ i(∩v)▼</td>
<td>[∨, ) ] =</td>
<td>REDUCE error and recovery ②</td>
</tr>
<tr>
<td>▲ C ∧ ( C ∧ C ∧ ◆ ▼ )▼</td>
<td>[∨, ) ] =</td>
<td>REDUCE(C → C ∨ C)</td>
</tr>
<tr>
<td>▲ C ∧ ( C ∧ ◆ ▼ )▼</td>
<td>[ (, ] =</td>
<td>SHIFT</td>
</tr>
<tr>
<td>▲ C ∧ ( C ◆ ▼ )▼</td>
<td>(, ] =</td>
<td>REDUCE(C → (C))</td>
</tr>
<tr>
<td>▲ C ∧ ( C ◆ ▼ )▼</td>
<td>[ ] =</td>
<td>REDUCE(C → C ∧ C)</td>
</tr>
<tr>
<td>▲ C ∧ ( C ◆ ▼ )▼</td>
<td>[ ] =</td>
<td>REJECT because of errors ② and ②</td>
</tr>
</tbody>
</table>

Operator Precedence Parsing with Error-Recovery Routines
Lexical Analysis

- acceleration of the scanning process: scanning ahead on the input to recognize and buffer several next lexemes
- buffering these lexemes by using various economically data-organized methods (pairs of cyclic buffers)
- theory of finite automata
- minimization of the number of states in any deterministic finite automata

Syntax Analysis

- time and space complexity of parsing algorithms
- general parsers based upon tables
- Earley Parsing Algorithm
Topics Not Covered in This Book 2/4

Deterministic Top-Down Parsing
- $k$-symbol lookahead
- LL($k$) parsers based upon LL($k$) grammars
- automatic top-down parser generator

Deterministic Bottom-Up Parsing
- generalized precedence parser
- varies constructions of the LR tables and the corresponding LR parsers
- canonical LR parsers
- lookahead LR parsers
- the Brute-Force lookahead LR parsers
- shift-reduce and reduce-reduce problems discussed in detail
Topics Not Covered in This Book 3/4

Syntax-Directed Translation and Intermediate Code Generation
- top-down syntax-directed translation discussed in detail
- semantic pushdown
- stack-implemented tree-structured and hash-structured symbol tables
- more software tools, such as bison

Optimization and Target Code Generation
- time and space complexity
- optimizing compiler
- run-time memory management
- static memory management
- dynamic memory management
- stack storage and heap storage
Topics Not Covered in This Book 4/4

**Theory**
- deterministic parsers of non-context-free languages
- conditional grammars
- regulated grammars

**Design**
- compiler design based upon computational cooperation, distribution, concurrence, and parallelism
- functional, logic, and object-oriented languages and their compilers
Discussion and End